Value types

- Value types
  - Struct
    - Mutable struct problem
  - Nullable
  - Guid
  - Dates, times
    - DateTime
    - TimeSpan
    - DateTimeOffset
  - Enum
    - Enum Flags

Struct

Структуры позволяют создавать пользовательские значимые типы.

```
struct Example
{
    public int Value;
    public string SomeString; // Можно указывать ссылочные типы

    public string ExampleMethod()
    {
        return $"{value} - {SomeString}";
    }
}

Example e = new Example();
e.Value = 1;
e.SomeString = "xmpl";

Console.WriteLine(e.ExampleMethod()); // 1 - xmpl
```

Ключевые особенности:

- Могут:
  - включать конструкторы, константы, поля, методы, свойства, события, операторы, вложенные типы
  - реализовывать интерфейс
- Не могут:
  - наследоваться от другой структуры или класса
  - выступать в качестве базового класса
  - содержать конструктор без параметров
- При присваивании к новой переменной создается копия объекта. Все изменения в новой копии не влияют на старую.

Рекомендации от Рихтера по создаю своего value type:

- малый размер - до 16 байт
- ведет себя как примитивный: в частности immutable поведение, отсутсвие методов изменяющих состояние полей, по сути readonly
- тип не имеет базового и производных от него

```
struct Vector
{
    public readonly double X;
    public readonly double Y;

    public Vector(double x, double y)
    {
        X = x;
        Y = y;
    }
}
...
var vector = new Vector(5, 3);
```

Mutable struct problem

Почему делать структуру изменяемой плохая идея?

- С одной стороны она будет иметь mutable поведение в некотором методе
- В других ситуациях фреймворк будет делать для нее immutable поведения, как для структуры:
  - например, при передаче в качестве параметра или при присвоении
- При этом некоторые передачи параметров внутри фреймворка так законспирированы, что программист не подозревает об их существовании
- Все оптимизации во фреймворке расчитаны на то, что все значимые типы immutable.

Использование mutable структур ведет к коду, логика работы которого абсолютно неочевидна

Вывод: **никогда не делайте Mutable структуры** (из исключений - спец. оптимизиация внутри фреймворка, см. енумератор)

К каким ошибкам это приводит раз, два

Readonly struct

В C#7 добавили возможность красиво сделать immutable структуру:

- все публичные поля должны быть readonly
- все изменения полей /this только в конструкторах

```
public readonly struct S
{
    public int Age { get; set; } // низя

    public S(int age) { this.Age = age; }
    public S(S other) { this = other; }

    public S Replace(S other)
    {
        S value = this;
        this = other; // низя
        return value;
    }
}
```

Оптимизации структур

- Можно использовать `ref` / `out` / `in` атрибуты для передачи параметров в методы

В C#7 улучшили возможности оптимизации valuetype:

- `ref struct` тип
  - Тип может быть истанцирован только на стеке
  - Нельзя использовать как член класса или структуры, не может boxing/unboxing, не может быть static, нельзя использовать в async методах
  - Можно использовать только для передачи в методы / local variable
- `ref return`
- Можно одновременно использовать `readonly ref struct`
- Добавили новые типы `Span<T>` and `Memory<T>` и др.

Nullable

Ссылочным типам можно присваивать `null`. Ссылка будет содержать все 0. Никаких специальных полей для `null` нет.

Поэтому достаточно очевидно, что значимым полям, которые хранят только значение нельзя присвоить `null`.
В C# 1 нельзя было присваивать `null` значимым типам вообще и предлагалось всячески изощряться.

В C# 2 для значимых типов ввели конструкцию nullable, которая позволяет присвоить null.
Это делается через системную структуру `System.Nullable<T>`:

```
int? x = 10;
System.Nullable<int> x = 10; // Эквивалентные записи

Guid? y = null;
```

Упрощенный вид `System.Nullable<T>`:

```
public struct Nullable<T> where T : struct
{
    private Boolean hasValue = false; // По дефолту null
    internal T value = default(T);    // По дефолту все биты обнулены

    public Nullable(T value)
    {
        this.value = value;
        this.hasValue = true; // Выставляем, что есть значение
    }
```

Свойства и методы:

```
// Есть ли значение?
public Boolean HasValue { get { return hasValue; } }

// Значение, бросаем исключение если идет доступ к элементу, когда его нет
public T Value
{
  get
  {
    if (!hasValue)
    {
      throw new InvalidOperationException("Nullable object must have a value.");
    }
    return value;
  }
}
```

Еще методы:

```
// Получение Value или дефолтного значения
public T GetValueOrDefault() { return value; }
public T GetValueOrDefault(T defaultValue)
{
    if (!HasValue)
        return defaultValue;
    return value;
}

public override Boolean Equals(Object other)
{
  if (!HasValue)
    return (other == null); // Если оба объекта null, то они равны

  if (other == null)
    return false;

  return value.Equals(other);
}
```

```
public override int GetHashCode()
{
    if (!HasValue)
        return 0;
    return value.GetHashCode();
}
public override string ToString()
{
    if (!HasValue)
        return "";
    return value.ToString();
}
public static implicit operator Nullable<T>(T value)
{
    return new Nullable<T>(value);
}
public static explicit operator T(Nullable<T> value)
{
    return value.Value;
}
```

Пример использования nullable:

```
int? i = 6;
Console.WriteLine($"{ i.Value } { i.HasValue }");  // 6 true

int x = (int)i; // Явное приведение к обычному int
int? y = x;     // Неявное приведение от int
i++;            // i = 7 Можно выполнять операции
i = null;
Console.WriteLine($"{ i.Value } { i.HasValue }");    // false

if (i == null) {}
if (i.HasValue)
{
    int k = i.Value;
}
if (i == y) {}
```

- Интересно, что при приведении `Nullable` к ссылочному типу (`object`, например):
  - если `HasValue == false`, то результирующая ссылка равна `null`. Так сделано для того, чтобы проверка на null оставалась верной вне зависимости от того, упаковано значение или нет
  - иначе значение упаковывается, как если бы он не был nullable, т.е. `((int?)10).GetType() == 10.GetType()`.
- Сделать `Nullable<Nullable<int>>` нельзя 😃
- К nullable типу можно применять стандартные операторы (`+`, `*`, `>`, `^`, `>>`, etc), но настоятельно рекомендуется этого не делать и обрабатывать ситуацию `if (!i.HasValue)` отдельно
- `==` при компиляции превращается в обращение к `.HasValue`, так что оба варианта проверки являются эквивалентными
- Если один из операндов равен `null`, то результат будет null/false в большинстве операций, но есть моменты
  - `==`, `!=` - если оба операнда `null`, то они считаются равными
  - `null | true` вернет `true`
- Надо ли рассказывать про хелпер System.Nullable ?
  - `public static int Compare<T>(Nullable<T> n1, Nullable<T> n2)`
  - `public static bool Equals<T>(Nullable<T> n1, Nullable<T> n2)`
  - `public static Type GetUnderlyingType(Type nullableType)`

Guid

Guid - global unique identifier - часто используемая в бд структура.

- 16 байт.
- Есть несколько версий того, как его генерить, раньше MS генерило по Mac-адресу сетевой карты, текущей дате, но вроде как это было небезопасно. Сейчас в mssql генерится на основании рандома. Как в c# сейчас не в курсе.
- Обеспечивает глобальную уникальность сущности, вероятность повторения очень-очень мала, в духе 50% вероятности коллизии, если генерить миллиард записей в секунду, 45 лет подряд.
- `623ab58a-afc4-46c8-820e-c0a0686c1d90` каноническое строковое представление, разделенное по 8-4-4-4-12 символов.

```
Guid value = Guid.NewGuid();    // Генерация нового значения
value = Guid.Empty; // Зарезервированное значение по-умолчанию (со всеми нулями)

value = Guid.Parse("c5d370a0-55d9-445a-b3d6-a2df47d2f233");
byte[] byteArray = value.ToByteArray();  // 16 byte array
```

Pros:

- Позволяет генерить идентификаторы для базы данных на клиенте, особенно востребовано в шардированных базах.
- Уникален и на уровне таблиц и бд и серверов
- Позволяет легко мерджить данные
- В репликации используются
- Несколько более безопасен для ссылок, но всё равно `Do not assume that UUIDs are hard to guess; they should not be used as security capabilities` Guid RFC

Cons:

- В 2 раза больше bigint
- Плох в качестве кластерного индекса, потому что:
  - большой
  - не последователен (для борьбы с этим в Sql Server есть функция генерации упорядоченных `guid`-ов):
    - вставка в середину ведет к фрагментации и постоянной перестройке индекса,
    - "последовательные" данные не локальны (random read vs sequential read).
- Нечитаем

Dates, times

MSDN Работа со временем

- `DateTime` - дата, время и двухбитовое поле (Kind)
- `TimeSpan` - интервалы времени
- `DateTimeOffset` - локальные дата и время + смещение локального времени относительно UTC.
- `TimeZone` - класс для работы с зонами, конвертации времени между ними (выходит за пределы курса)

MSDN Choosing article

DateTime

DateTime - структуря для работы с датой и временем

Время измеряется в отрезках по 100 наносекунд, которые называют `ticks`.
64 bit: 62 содержат ticks, остальные 2 bit содержат поле enum `Kind`, которое определяет "тип" даты:

- `Unspecified` - время без указания timezone
- `Local` - локальное время со смещением от utc и возможным переходом на летнее время
- `Utc`

```
DateTime date0 = new DateTime();          // минимальное время
DateTime date1 = new DateTime(2017,10,3);// 03.10.2017 0:00:00 Unspecified
DateTime utcTime = new DateTime(2010, 11, 18, 17, 30, 0, DateTimeKind.Utc);
DateTime date2 = DateTime.MinValue;       // 01.01.0001 0:00:00
DateTime date3 = DateTime.MaxValue;       // 31.12.9999 23:59:59
DateTime date4 = DateTime.Now;            // Kind == Local
DateTime date5 = DateTime.UtcNow;         // Kind == Utc
DateTime date6 = DateTime.Today;
DateTimeKind kind = date5.Kind;           // DateTimeKind.Utc
var value = date1.AddHours(3);
```

Отображение и разбор даты из строки очень сильно зависит от региональных стандартов.

```
DateTime d = DateTime.Parse("03.10.2017 13:45:43"); // 03.10.2017 13:45:43
d = DateTime.Parse("5/1/2008 8:30:52 AM",
System.Globalization.CultureInfo.InvariantCulture); // 01.05.2008 8:30:52
```

CultureInfo - информация о специфической культуре в c#

```
// Культура текущего потока, используется в дефолтном парсинге/выводе
CultureInfo currentThreadCulture =
System.Globalization.CultureInfo.CurrentCulture;

// Культура, которую используется ResourceManager при подстановке правильных
ресурсов
CultureInfo cultureForResourceManager =
System.Globalization.CultureInfo.CurrentUICulture;

var newCulture = new CultureInfo("ru-RU");
System.Globalization.CultureInfo.CurrentCulture = newCulture;
```

```
DateTime now = DateTime.Now;
string[] formats = new string[] {"D", "d", "F","f", "G", "g", "M", "O", "R", "s",
"T", "t", "U", "u","Y"};
foreach(string s in formats) { Console.WriteLine($"{s}: { now.ToString(s) }"); }

Console.WriteLine($"{now:D}"); // Tuesday, 03 October 2017
/* D: Tuesday, 03 October 2017
d: 10/03/2017
F: Tuesday, 03 October 2017 01:39:28
f: Tuesday, 03 October 2017 01:39
G: 10/03/2017 01:39:28
g: 10/03/2017 01:39
M: October 03
O: 2017-10-03T01:39:28.5397283+03:00
R: Tue, 03 Oct 2017 01:39:28 GMT
s: 2017-10-03T01:39:28
T: 01:39:28
t: 01:39
U: Monday, 02 October 2017 22:39:28
u: 2017-10-03 01:39:28Z
Y: 2017 October */
```

Формат можно задать более конкретно:

```
DateTime now = DateTime.Now;
Console.WriteLine(now.ToString("hh:mm:ss"));    // 13:05:55
Console.WriteLine(now.ToString("dd.MM.yyyy"));  // 05.01.2008
```

Особенности:

- Допустим, вы получили дату как `DateTime.Now` (Local), сохранили ее в бд, прочитали оттуда Unspecified. Это плохо.
- По-хорошему надо сравнивать DateTime только с одним `Kind` (при сравнении DateTime kind не учитывается)

TimeSpan

TimeSpan - Структура для хранения интервалов времени

```
DateTime date1 = new DateTime(2010, 1, 1, 8, 0, 15);
DateTime date2 = new DateTime(2010, 8, 18, 13, 30, 30);

TimeSpan interval = date2 - date1;

Console.WriteLine("{0} - {1} = {2}", date2, date1, interval.ToString());
Console.WriteLine($"{interval.Days} {interval.TotalDays} {interval.Hours}");

TimeSpan zeroTimeSpan = TimeSpan.Zero;
interval = interval + TimeSpan.FromDays(10);
TimeSpan value = new TimeSpan(4, 0, 0); // 4 часа
```

DateTimeOffset

DateTimeOffset - структура для хранения DateTime вместе со смещением от UTC.

- Содержит абсолютное время (впрочем, как и DateTime с kind==utc)
- В отличие от DateTime содержит информацию и об абсолютном времени, и о локальном (смещение Offset)
- Не включает `Kind` поле
- Содержит такую же по формату дату, как DateTime
- Надо понимать, что даты и смещения недостаточно, чтобы полностью сохранить информацию о TimeZone пользователя. Смещение не позволяет корректно идентифицировать TimeZone пользователя, ведь не только несколько временных зон могут обладать одним смещением, но и смещение одной зоны может меняться от перехода на летнее время.
- Если конвертить DateTime в DateTimeOffset (а это происходит неявно), то `DateTime.Kind` важен:
    - При utc будет просто нулевой offset
    - При unspecified/local - она будет использовать текущий local для конвертации. Очень небезопасно!

Методы практически повторяют `DateTime`:

```
DateTimeOffset dateOffset1 = DateTimeOffset.Now;
DateTimeOffset dateOffset2 = DateTimeOffset.UtcNow;

TimeSpan difference = dateOffset1 - dateOffset2;
TimeSpan offset = dateOffset1.Offset;  // Offset - это TimeSpan!
```

Советы/замечания:

- Либо используйте `DateTimeOffset`, либо используйте только DateTime с `DateTimeKind.Utc` везде (особенно сохранение в бд)
- Если вы хотите сохранить момент времени, в который выполнялось действие, как его видел пользователь, вы **обязаны** использовать `DateTimeOffset`
- Если вы хотите модифицировать ранее прихраненный `DateTimeOffset`, то его Offset может поменяться и надо прихранивать `TimeZone.Id`
- `DateTime` хорошо использовать для:
  - только дата
  - только время
  - общие штуки, когда смещение не нужно (будильник сделать на определенное время)
- `DateTimeOffset`
  - однозначный момент во времени
  - общая арифметика с датами и временем

MSDN:

> These uses for DateTimeOffset values are much more common than those for DateTime values. As a result, DateTimeOffset should be considered the default date and time type for application development.

На самом деле для серьезной работы со временем не подходит/неудобен ни один из встроенных типов. В таких случаях лучше воспользоваться специализированными библиотеками типа `NodaTime`. И вот почему

Enum

Enum - Перечисление - набор связанных пар, состоящих из строки и целочисленного значения (int / byte / short / long, по-дефолту `int`).

Цепочка наследования `System.Object` -> `System.ValueType` -> `System.Enum` -> UserDefined Enum

```
enum Color  // Минималистичная форма записи
{
    Red,    // Компилятор выставит соответствие 0
    Green,  // 1
    Blue    // 2
}

Color myVariable = Color.Green;
Console.WriteLine(myVariable);  // Green

int i = (int) myVariable; // Явно приводится к целочисленному типу
Console.WriteLine(i);     // 1
Color value = (Color) (i + 1);  // И обратно приводится
Console.WriteLine(value);       // Blue
```

Всегда задавайте все значения енама вручную, чтобы облегчить поддержку кода.

Аналогичный предыдущему результат:

```
enum Color : int
{
    Red = 0,
    Green = 1,
    Blue = 2
}
```

Компилируется примерно в такую структуру (мы, конечно, не можем сами написать такой код, унаследоваться от enum нельзя):

```
struct Color : System.Enum
{
    public const Color Red = (Color) 0;
    public const Color Green = (Color) 1;
    public const Color Blue = (Color) 2;

    public Int32 value__; // Нельзя обращаться напрямую
}
```

Значение по-умолчанию для первого элемента 0, потом инкремент от предыдущего:

```
public enum Color
{
    Red,
    Green = 4,
    Blue
}

public static void Main()
{
    foreach(var color in Enum.GetValues(typeof(Color)))
    {
        Console.WriteLine($"{color} - {(int)color}");
    }
    // Red - 0
    // Green - 4
    // Blue - 5
}
```

Посмотрим, что будет, если отрицательное значение зафигачить:

```
public enum Color
{
    Red,
    Green = -1,
    Blue
}

public static void Main()
{
    foreach(var color in Enum.GetValues(typeof(Color)))
    {
        Console.WriteLine($"{color} - {(int)color}");
    }
    // Red - 0
    // Red - 0
    // Green - -1
}
```

Поэтому рекомендация: всегда явно указывать значения всех элементов

Возможны невалидые состояния:

- Мы можем сконвертить к enum любой int и это не вызывет ошибки.
- Значение по-умолчанию для enum `0` и даже, если вы не задали такого элемента, то clr всё равно выставит дефолтом `0`.

```
public enum Color
{
    Red = 2,
    Green = 3,
    Blue = 4
}
public static void Main()
{
    Color c = new Color();
    Console.WriteLine($"{c} - {(int)c}"); // 0 - 0
    c = (Color) (-1); // -1 - -1
    bool flag = Enum.IsDefined(typeof(Color), c); // False
    c = (Color) 2; // Red - 2
}
```

Основные методы для работы с enum:

- GetValues / GetNames
- Parse / TryParse
- IsDefined - Проверяет допустимость числового значения для енама, работает через reflection, то есть медленно. В него можно пихать как значения типа, так и строки, и он всегда работает со строками с учетом регистра

```
Color[] values = (Color[]) Enum.GetValues(typeof(Color));
string[] names = Enum.GetNames(typeof(Color));
Color value = (Color) Enum.Parse(typeof(Color), "Green");

Object Parse(Type enumType, String value);
Object Parse(Type enumType, String value, Boolean ignoreCase);
Boolean TryParse<TEnum>(String value, out TEnum result);
Boolean TryParse<TEnum>(String value, Boolean ignoreCase, out TEnum result);
Boolean IsDefined(Type enumType, Object value);
```

Посмотрим, как работает парсинг из строки. Поиграем в игру, угадай как работает framework:

```
public enum Status
{
    New = 2,
    Commited = 3,
    Deleted = 4
}

string[] test = new []
{
    "New",
    "new",
    "3",
    "0",
    "",
    "-1",
    "New, Commited"
};
```

Для каждой строки будем находить:

```
Status value = (Status) Enum.Parse(typeof(Status), s);
int intValue = (int)value;
bool isDefined = Enum.IsDefined(typeof(Status), value);
```

Полный итоговый код с результатами:

```
public enum Status
{
    New = 2,
    Commited = 3,
    Deleted = 4
}

public static void Main()
{
    string[] test = new []
    {
        "New",  // New | 2 | True
        "new",  // Exception: Requested value 'new' was not found.
        "3",    // Commited | 3 | True
        "0",    // 0 | 0 | False
        "",     // Exception: Must specify valid information...
        "-1",   // -1 | -1 | False
        "New, Commited" // Commited | 3 | True      ~WTF~LUL~
    };
```

```
    foreach (string s in test)
    {
        try
        {
            Status value = (Status) Enum.Parse(typeof(Status), s);
            int intValue = (int)value;
            bool isDefined = Enum.IsDefined(typeof(Status), value);

            Console.WriteLine($"`{s}`: {value} | {intValue} | {isDefined}");
        }
        catch(Exception e)
        {
            Console.WriteLine($"`{s}`, Exception: {e.Message}");
        }
    }

    Enum.IsDefined(typeof(Status), "New, Commited"); // false
}
```

В связи со всем этим рекомендуют:

- Создавать отдельный элемент Unknown / None / Default = 0 в enume
- Пользоваться методом `Enum.IsDefined` для проверки валиднсти значения enum
- Не создавать элементы enum "на будущее"
- Вообще enum в C# выглядит глобально поломанным, можно пробовать использовать сторонние реализации, например, EnumNet

Enum Flags

Можно сделать енам, используемый для идентификации битовых флагов.

```
[Flags]     // Надо пометить класс аттрибутом Flags
enum Actions
{
    None = 0,       // Можно сделать дефолтное значение
    Read = 0x0001,  // Обязательно нужно проставить всем значения
    Write = 0x0002, // Обычно устанавливают значимым лишь один бит (то есть
назначают степени двойки значениями)
    ReadWrite = Actions.Read | Actions.Write,
    Delete = 0x0004,
    Query = 0x0008,
    All = 0x000F,   // Можно назначить не степень двойки, тогда ToString вернет
All
}

Actions actions = Actions.Read | Actions.Delete; // 0x0005
Console.WriteLine(actions.ToString());           // "Read, Delete"
```

Как выглядит в коде примерная работа с битовыми флагами (вариант без проверок):

```
bool IsSet(Actions value, Actions valueToTest)
{
    return (value & valueToTest) == valueToTest;
}

bool IsClear(Actions value, Actions flagToTest)
{
    return !IsSet(value, flagToTest);
}

Actions Set(Actions value, Actions setFlags)
{
    return value | setFlags;
}

Actions Clear(Actions flags, Actions clearFlags)
{
    return flags & ~clearFlags;
}
```

- Методы, описанные ранее, работают и с битовыми флагами
- `IsDefined` не работает правильно с битовыми флагами! Его форма работы со строками не рассчитана на запятые (всегда возвращает false).
- `ToString`, если нашел `[Flags]`, то рассматривает перечисление, как набор битовых флагов